CS 105 Perl: Regular Expressions

Nathan Clement

February 5, 2014
Agenda

We will cover regular expressions on a basic level, with a basis in automata theory. Then we’ll delve into some simple Perl regular expression syntax.

- Finite Automata
- Regular Languages
- Regular Expressions
- Atoms
- Metacharacters
- Character Classes
Documentation

For more documentation on regular expressions, please see:

- `perlrrequick` - quick start
- `perlretut` - tutorial
- `perlreref` - quick reference
- `perlre` - full reference
Finite Automata

The terms *finite automata* and *finite state machine* are roughly equivalent. The former, however, has a precise mathematical meaning, whereas the latter is more generic.

- Finite
- Automaton (5-tuple)
  - States
  - Alphabet
  - Transitions
  - Start state
  - Accept State(s)
- Language
Simple Finite Automaton

\[
\text{start} \quad a \quad b \quad b \quad b \\
\quad b \quad a \quad a \\
\quad a \\
\quad \text{accept state}
\]

input seen:
ab
More Useful Finite Automaton

The diagram illustrates a finite automaton with states labeled as follows:

- **start**
- **firstDigit**
- **decimalPt**
- **moreDigits**
- **finalDigits**

Transitions are labeled with character sets that denote what symbols the automaton can read to transition between states:

- **start** with transitions to **firstDigit** on "+" and "-".
- **firstDigit** with transitions to **moreDigits** on "+" and "-".
- **moreDigits** with transitions to **finalDigits** on "+" and "-".
- **decimalPt** with transitions to **finalDigits** on "+" and "-".

Character sets include:

- **[1-9]**
- **[0-9]**
- **.**

These symbols represent the input symbols that the automaton can accept to transition through its states.
Regular Languages and Expressions

• A language is *regular* if it can be detected by a finite automata
• A finite automata can only match regular languages
• A *regular expression* is a way to express ("write down") a finite automata
  • and thus can recognize regular languages
Fundamental Constructions for Regular Languages

• Regular languages are “closed” under the following operations:
  • Concatenation
  • Union
  • Kleene star

• In other words, all of these result in another regular language:
  • The concatenation of two regular languages
  • The union of two regular languages
  • The Kleene star of a regular language
Concatenation Example

concatenation
Union Example

union

minimize
Kleene star

\[ a^* \]

Same as

infinite union
Kleene star explanation

• The Kleene star basically means you can repeat a regular language any number of times (including zero) and the resulting set of languages is still regular, even though the set is infinite in size.
Matching a Number

• Remember the complex finite automaton from the beginning of the language? Here it is as a regular expression (complete with Perl syntax):

```
/\[+-\]?[1-9][0-9]*\.(\.[0-9]*)?\/?
```

Assembling regular expressions

• Recall that regular languages are closed under concatenation, union, and Kleene star.
• In regular expressions, we might also use these terms:
  • Juxtaposition: \texttt{ab}
  • Alternation: \texttt{a | b}
  • Repitition: \texttt{a*}
• This is the real Perl syntax. Use juxtaposition for concatenation, alternation with the pipe operator for union, and the asterisk (star) for the Kleene star.
Atoms

• We can build regular expressions from atoms. There are a number of atoms we use in Perl regular expressions:
  • Literal characters
  • Escaped double-quotable characters (\n, \t, etc.)
  • . ("anything" wildcard)
  • Character classes
  • etc, etc (this is not an exhaustive list)
Example
Example

• Juxtaposition

/foo/
Example

• Juxtaposition
  
  /foo/

• Alternation
  
  /foo|bar/
Example

• Juxtaposition
  `/foo/`

• Alternation
  `/foo|bar/`

• Repetition
  `/foo*/ # probably not what you want`
Repetition Intuitively

• Think of * as zero or more
• As in the previous star, one or more is probably what you really want.

/o*/  # zero or more o’s
/o+/  # one or more o’s
/oo*/  # same as /o+/  
• *, +, and other operators that express different types of repetition are called quantifiers
Precedence in Regular Expressions

• Precedence from high to low:
  • Repetition: *, +, other quantifiers
  • Juxtaposition
  • Alternation

While the precedence rules of regular expressions are often ignored, they are very important for understanding how your regular expressions work, and why certain constructs are often mistakes.
Precedence Example

/foo*/    # match fo, foo, fooo, etc.
# vs.
/(/foo)*/ # match '', foo, foofoo, etc.

Parens are used for grouping, but in this case, we can intuitively think of them as a way to override precedence (just like in math and most programming languages, i.e. they have the highest precedence).
Another Quantifier

• Already discussed the * and + quantifiers
• Sometimes you want to express something as optional, i.e. you might not see it, or you might see it one time only. For this, we would use ?.

/o*/ # zero or more o’s
/o+/ # one or more o’s
/o?/ # zero or one o
Anchors

• In the previous examples, such as /foo/, the string foo might be found and successfully matched anywhere in the string. Sometimes we want to only successfully match at a given position within a string.

/^foo/  # match foo only at the beginning
/foo$/  # match foo only at the end
    # (or before newline at the end)

• The ^ and $ hold the matching to a particular position that (intuitively) doesn’t move, which is why we call such a metacharacter an anchor.

• Special note: ^ and $ do not match characters.
Zero-width assertions

- Recall from the previous slide that \(^\text{^}\) and \(\text{\$}\) don’t match characters, but rather positions that technically exist between characters.
- When we want to express a matching condition between characters rather than on the characters themselves, we call this a zero-width assertion. \(^\text{^}\) and \(\text{\$}\) belong to a particular category of zero-width assertion that we call “anchors.” Typically all anchors are zero-width assertions, but not all zero-width assertions are anchors.
Another Precedence Example

/^foo|bar$/
# vs.
/^\(foo|bar\)$/

- These two regular expressions recognize two very different languages.
Another precedence example (graphically)

```
/^foo|bar$/
```

```
/^((foo|bar)$)/
```
Metacharacters

• A metacharacter is any character in a regular expression that has a special meaning and so isn’t interpreted literally.

• Here are some of the metacharacters we’ve seen so far: | (alternation)
  • *, +, and ? (quantifiers)
  • .
  • ^, $ (anchors)
  • ( and ) (grouping)
  • [ and ] (character class)

If you need to match one of these characters, simply escape it by putting a backslash (\) in front of it.
Better Atoms - Character Classes

• Sometimes you would like to match any of a certain kind of character, such as letters or numbers. Doing an alternation would be very tedious.

/^\d$/ # Never do this
/^\d$/ # Better; still not good
/^\d$/ # Decent

• A character class just lists all the valid characters that it can match. The characters must be delimited by square brackets [ ].
Exclusionary Character Classes

- Sometimes you want to match anything but certain characters.
- Put a caret (^) as the first character inside the square brackets. This makes the character class invert the sense of the match: “anything **not** listed is accepted”, rather than “anything listed is accepted.”

/[^0-9]/ # Match anything except a digit
Using - and ^ Inside Character Classes

• Obviously, since – and ^ have special meaning for character classes, they are a kind of metacharacter.

• A caret will have its normal (literal) meaning if it is not the first character in the character class, or you can escape it with a backslash.

• A hyphen only has a special meaning if it’s between two characters. If placed at the beginning or end of the character class, it will have its normal (literal) meaning. Or, again, you can escape it with a backslash.

/[^^-]/ # anything except a caret or hyphen
Handy Character Classes

• Recall how I claimed `/^[0-9]$/` was merely “decent.”

• This is because Perl already includes a character class for digits.

```
/^\d$/   # Excellent!
```

• `\d` is the character class for digits; `\D` is the character class for non-digits.
More Handy Character Classes

`//d/` # digit

`//D/` # non-digit

`//s/` # whitespace (space, tab, newline)

`//S/` # non-whitespace

`//w/` # word, or `[A-Za-z0-9_]`

`//W/` # non-word